Report No. 91004

Report No. 91004

ROYAL SIGNALS AND RADAR ESTABLISHMENT,
MALVERN

FROM ENGLISH INTO Z, A REVERSE SPECIFICATION PROCESS

Author: S C Giess

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

June 1991

CONDITIONS OF RELEASE

0107583 304189

********** DRIC U

********** DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report 91004

Title: From English into Z, a reverse specification process.

Author: S. C. Giess

Date: June 1991

Abstract

On an exercise to translate the communication protocol specification RFC 826, 'The address resolution protocol', already written in English, into the mathematical specification language Z.

Contents

Introduction

Part of the work programme of the Distributed Information Systems Division of the DRA at Malvern is to create approaches to the specification of communication protocols which use formal languages that are powerful enough to capture all the necessary actions of these systems. Such languages aim to allow reasoning concerning the operation of the protocols with the goal of minimising if not completely avoiding design and implementation problems. Such problems can arise because these protocols are frequently defined by means of documentation written in English.

The use of English descriptions, when applied to the specification of large systems, is well known to have dangers. These are (i) differences in meanings attached to English words between the writer of the document and the reader of the document and (ii) the possibility of the English description being incomplete. This incompleteness could be due to the designer having an environment in mind of which the reader is unaware. Alternatively the designer may have originally decided that another module of the system will handle some conditions but then forgot to incorporate them in the other module due to the development time-span (It can often take several years to develop a communications protocol). Of course the incompleteness could just be due to simple oversight.

These dangers can be reduced by using a notation that compactly expresses the desired functionality. If the notation has a mathematical foundation then the possibility of detecting inconsistencies and omitted parts in such specifications is further increased. However there is a penalty in the need for the designer to understand aspects of pure mathematics for the full correct use of such notations. Whether this penalty will turn out to be a serious problem will, I suspect, only become clear after a large body of designers whose first discipline was not Pure Mathematics have tried such notations on real world problems.

This report investigates these problems by setting out to specify an existing communications protocol in a formally based language.

There are several formal languages extant which are comparatively mature in terms of syntax and tool support, eg VDM, **Z**. There are others which are specifically designed for protocols, eg LOTOS, Estelle, however they and their tools are still in varying stages of development and do not have the same reasoning potential nor can they handle timing questions. There are others which are just starting to appear from the research phase which have the reasoning potential, eg CSP [1] and can handle questions of timings, eg Timed CSP [2].

It was considered that an attempt to specify a real protocol in Z [3] could be a useful step in the exploration of the issues mentioned above to the mutual benefit of the protocol designers, the protocol implementors and formal language creators. The particular protocol was chosen because it was large enough to be non-trivially useful in exploring the issues, yet small enough not to be overwhelming. Moreover it had already been implemented by other members of the Distributed Information Systems Division.

It is hoped that the work described here will show protocol users the benefits to be obtained from formality.

The English description of the specification

The Network Working Group's Request for Comments No. 826 (RFC 826) describes, in informal English, the elements of an Ethernet address resolution module. In November 1982 a draft version of this Protocol was circulated for comments among the Network Working Group (NWG) of the ARPA internet community. This draft has subsequently become the de facto definition [4] and is the one considered in this memorandum (see appendix A). It is worth noting that this protocol is a small part of an extensive suite of protocols which, when taken together, constitutes the operational basis for one of the largest communication communities in existence.

The context for this address resolution protocol is as follows: Consider a Local Area Network (LAN) comprising several host systems connected by an Ethernet cable. Some of these LAN members are also connected to other networks, with the packet data transfer being controlled by ARPAnet software. Some of the LAN members are also running ARPAnet software and wish to send or receive data to other systems also running ARPAnet software perhaps on other networks. However the addressing scheme used by the Ethernet system is not the same as that used by the ARPAnet. So, since the LAN requires Ethernet addresses, it is neccessary to find the Ethernet address of the desired machine (if it happens to be on the same LAN) or the Ethernet address of a LAN member that can route the data onwards. This requires the existence of a software module to effect the address translation in each LAN member running ARPAnet software. The ARPA community calls this action "address resolution".

We see that the purpose of the module is to return the appropriate Ethernet address when given a protocol type and an address in that protocol's format. The module does this by storing the information in an address resolution table. When it is asked for the Ethernet address of another address type it first examines this table to see if the information is already present. If it is then the Ethernet address is returned. If the information is absent then the module broadcasts a request to all other units on the Ethernet to see if any of them is the wanted unit. If the

wanted unit is present it sends a reply to the requestor and at the same time, updates its own table with the requestor's address information. When the requestor receives a reply it updates its own table.

The informal specification given in RFC 826 is almost an implementation algorithm recipe of this process. However we shall see that it makes assumptions which rely on the implementor's knowledge of the operation of the rest of the system.

Modular decomposition of the specification

There are two viewpoints to the description/specification of a protocol:

The first, the "global" view, considers the definition of the operation of the protocol as a whole, ie the interactions that all the relevant nodes have to undertake for the desired result to be achieved. This is the viewpoint of the designer of the protocol.

The second, the "local" view, considers the definition of the constituent processes that a particular node should be able to do in order to carry out the protocol. This view is typically that which the implementor needs when writing software for a particular host.

Clearly these two viewpoints on the protocol have to be consistent for the protocol to work in practice. Checking that this is indeed so is an important part of verification.

One advantage gained by using a formal language is that it is easier to prove that "local" view specifications are in full agreement with the "global" specification. However we must remember that the decomposition itself of the global into the local may require validation. This point is illustrated by RFC 826 which is a small part of a larger system. This is because the specification, as written, is not complete, relying upon other parts of the protocol suite for correct operation. Hence the change of viewpoint from global to local is an implementation design decision, which in turn may affect some system goal on the part of the protocol designer not explicitly stated in the RFC document.

The global view

In order to understand the operation of the protocol we study the English definition given in appendix A from the global viewpoint, decomposing its overall operation into a collection of component processes whose individual specification requirements can be easily captured. The reader is well advised to study appendix A carefully. In particular observe what the English actually says, as compared to what the reader may be expecting to see by virtue of any prior personal experience with communications protocols.

This decomposition is done from the starting point of a node which has had a request for address information passed down to it from a higher protocol level.

First we define the operation of the protocol as a whole, calling it process module Address_protocol. This process has to either return the Ethernet address from its own data table, ie a local search, or update its own data table by inquiring of the other systems on the Ethernet, a remote search.

We can represent the local search process by the single module Successful_local_find and the remote search by the module Successful_table_update. So the global operation is an OR expression between a successful local search and a successful remote search, that is:

Address_protocol = Successful_local_find $\vee$ Successful_table_update

where the symbol $\vee$ is used informally to indicate a choice.

However the remote search process itself can be decomposed into a sequence of actions comprising six stages. In the first stage the request for information is despatched to other members on the Ethernet - called module Send_request. In the second the request is received by the other members - module Receive_request. Here each member checks for the applicability of the request to it. The third stage checks whether the requestor is known to it - module Existing_entry_check. If it is not, then, in the fourth the requestor's data is added to node's table, provided the node is the one wanted - module Add_new_entry. The fifth stage sends a reply to the requestor - module Reply. Finally, in the sixth stage, this reply is received by the originator which then updates its local table - module Update_table.

It can be seen that a successful remote search gives rise to a sequence of actions of the form:

Send_request $\rightarrow$ Receive_request $\rightarrow$ (Existing_entry_Check $\vee$ Add_new_entry) $\rightarrow$ Reply $\rightarrow$ Update_table.

Here the CSP symbol $\rightarrow$ is used informally to indicate the sequence of actions.

We note that this natural way of describing the protocol crosses machine boundaries, some processes are running on the originating local machine whilst others are on the remote machines as we show below:

<.... Overall .....> <..... Local .....

Address_protocol = Successful_local_find ⌄

..... Local .....> <..... Remote .....

(Send_request → Receive_request → (Existing_entry_Check ⌄

..... Remote .....> <.... Local .....>

Add_new_entry) → Reply → Update_table)

Here we have simplified the description of the remote process action to a single sequence. In reality it would be a parallel combination of distinct processes, one per LAN node.

The local view

Now the actual implementation in any one machine of the complete protocol has to cater for the machine acting in a local role or in a remote role. To achieve this the operation of the protocol can be considered to be three independent modules.

Informally, these are:

Address_module1 = Successful_local_find ⌄ Send_request

Address_module2 = Receive_request → (Existing_entry_check ⌄
Add_new_entry) → Reply

Address_module3 = Update_table

In Address_module1 the address resolution procedure has been passed a request for the address from the protocol level above it. This request is serviced by first mounting a search of the local address table. If the address is there then it is returned and that is the end of that procedure. If the search was unsuccessful then a request for information packet is created and broadcast to other machines on the Ethernet. This is also the end of the procedure. Hence the overall functionality is either a successful find or a successful send request, but not both.

In Address_module2 the address resolution procedure receives a request for information in the form of a packet sent over the Ethernet. The module consists of a sequence of tests which form the precondition to either an update of the table or the addition of a new entry, but not both. If any of the tests fails then the procedure terminates leaving the address table either unchanged or updated depending upon where in the chain the test failed.

In Address_module3 the address resolution procedure receives a reply to a request for information in the form of a packet sent over the Ethernet. The module checks that the packet is for it as a precondition and, if successful, updates the table otherwise the procedure terminates leaving the table unchanged.

The use of Z in writing specifications

A summary of the Z approach to writing a specification.

As a first step the complete system is decomposed, in a top down manner, into small units of modular activities. This is followed by the specification of each of these small activities by means of units called schemas. A schema explicitly states the constraints (predicates) which have to be met for the activity of a unit to be correct. It also explicitly defines the before - after relationship for relevant entities in the unit. Finally these individual schemas are combined to produce the configuration which is the specification of the whole system.

The underlying rationale is to ensure that the actions of the system are defined under all circumstances - both normal operation and error condition. In Z this concept is expressed by saying that the condition of the complete system, formed by the combination of the conditions of all the schema modules and their interconnection, is TRUE.

Now, if the predicate part of a schema is satisfied by an implementation then the process specified in the schema works succesfully. If the conditions are such that the predicate is false then, by convention, the implementation of that process can do anything. However if the schema is part of a schema expression which overall is true then the overall action is still defined. Ideally therefore we should define the protocol in Z in such a way that a failure of a part will still result in the state of the system being well defined. However we have chosen not to deal with the error conditions in this report. This decision has been taken so as to allow the issues arising from defining correct operation, the prime purpose of this report, to be clearly displayed and not masked by the syntatic structures needed to specify error condition handling.

This 'totality of possible circumstances coverage' is the central technique that formal methods uses to achieve reliable operation under all conditions. Nevertheless it should be remembered that sometimes real time constraints, like the cpu resources, mean that some of the desirable precondition tests may have to be omitted in the implementation, so resulting in weakness or holes in final system.

Hence what is desired cannot be achieved in a completely robust manner.

Finally it should be noted that Z has no concept of time. If it desired to specify a system in which decisions are made on the basis of time intervals, then Z cannot capture everything. Timed CSP may be a more appropriate route.

The Z specification

The Z specification of the modules described earlier is given in appendix B. From earlier it is apparent that the processes we are trying to specify are sequences of tests, together with operations which depend on the outcomes of these tests. An effective way to model such a chain in Z is to pass the information between the modules in the form of a message and apply the tests to the appropriate data fields. In this case a schema, having the structure of an Ethernet packet, is used as the message.

Comments on the specification process

First we note that in this paper we have attempted to provide an "after the event" specification - reverse engineering on two levels. The first level comprised taking the informal algorithm plus English explanation and constructing a global perspective description in order to understand the overall process. In the second level a specification, from the local perspective, suitable for an implementor had to be constructed from the global pespective. In both cases information from the implementor's "bottom up" algorithm was being used for the "top down" specifications.

It was necessary to have several discussions with the person who had implemented this protocol at RSRE in order to elucidate the software environment that the writers of the protocol clearly assumed. These were quite enlightening. We agreed that the informal specification had failings. I made these discoveries by attempting to follow the expected discipline of Z in being precise in matters when understanding the operation of the protocol. This had neccessitated the discussions to establish exactly what was the (unspecified) software environment. He made the discoveries when faced with the task of implementing the protocol with all the fine detail that an implementation requires. He said that he had had to make several implementation decisions in the absence of explicit detail in the specification. I found myself having to do the same in the Z specification with regard to the attempted generalisation to non-Ethernet hardware.

Specific points:

The first was that originally the protocol was only meant to produce the Ethernet address of an ARPA host running the Internet Protocol (IP). However in the drafting of the protocol there clearly had been a later desire to make it more flexible and handle other addressing regimes. However although this generalisation was attempted (by providing fields for different hardware spaces in the data part of the packet) it was not fully attained. This has resulted in a protocol that still assumes that

its packet traffic is only over an Ethernet, even though it claims generality in the specification's abstract. This inconsistency has been modelled in the Z specification by a precondition that requires the addresses (including Broadcast) in the packet header (called the Ethernet transmission layer) to be drawn from the Ethernet subset of possible hardware addresses. In addition the protocol address type of the sender has to be the same as that which is being sought. This also restricts its generality although it could be claimed that the purpose of the resolution module is only to set up addressing information for the specific protocol packet type in question.

The second point was that the testing of the validity of some of the operations was not as complete as may be desired. For example in the last stage of the protocol the update module receives an address resolution packet marked "reply". As written in the English description the module has no way of knowing whether the "reply" is to a request" actually originated by the the other half of the procedure in the same machine. The module just automatically updates the table This means that rogue packets or, worse, malevolent senders could disrupt the operations of the network by corrupting the address tables using syntactically correct but invalid "reply" packets. In a similar way the automatic updating of the table as defined in module Existing_entry_check would allow bogus "request" packets to corrupt the table entries.

The third point concerned what checks the module should make on the supplied data for consistency, eg checking that the format of the protocol address requested was indeed that of the protocol type declared. We have the issue here of what a module is expected to check explicitly and what not. The specification discusses some checks, however in general such decisions are left to the implementor.

The fourth point concerns failures. As described in the RFC document there appears to be no explicit failure action in the event of a failure of the update attempt. To wit, if the higher level service wishes to send a packet to an address that is not in the local table then the operations described in this document come into play. However if the update operation is unsuccessful then the higher level service is not informed in any way. This means that a subsequent request will be given the same treatment (ie as if the unknown address had never been looked for before) and not told that is has been looked for and not found. There is the chance here of an infinite loop unless the higher level service has some form of timeout. Now indeed there is a timeout but we are not told this fact in the RFC document. Hence from the point of view of the module specifier the specification is incomplete with regard to operational temporal stability.

General points:

Three general points are raised by this work. The first is that this protocol was written nearly ten years ago when formal methods of specifying systems were in their infancy. So the fact that flaws have been

found using these techniques demonstrates their usefulness as a design aid.

The second point concerns the role of a specification in the user-designer-implementor chain. I think it is fair to say that the ARPA protocols were written by designers who were also implementors, to be read by other peer designers who were also implementors. They used a style of documentation appropriate to that era and that situation of a concept under development. Unfortunately for others, they kept to the same style when it came to the final papers which defined the end protocols. Since those days Software Engineering practice has progressed. In particular the importance of writing a specification such that an implementation can be produced from it without the implementor having to know exactly where the entity specified fitted into the whole system. This software practice arises from common engineering practice where a designer of an entity expresses the design in the form of a plan, from which others can fabricate the entity. This practice also covers the situation where a later generation is tasked with implementing a specification, but who were not involved in its initial development. If this engineering approach were not to be followed then it would very difficult to embark on any enterprise whose size and complexity required teams of people working over several years with any hope of success.

These points have already been recognised and organisations like the CCITT are in the process of using the formal languages such as SDL [5] and ASN.1 [6] to specify future recommendations.

A third point arose out of comments from others during the work described in this paper. A position was made that the fact that this particular protocol had failings was not all that important as the protocol was merely an add-on to the main protocol suite, (not on the critical path for the operation of the Arpanet - unless of course an Ethernet connection was in involved, whereupon it would be on the critical path). Moreover the intention of the original Request for Comment was that the contents were a basis for experimentation and not a final definition. This is a valid position, however the act of promulgating this unaltered RFC in the DDN handbook meant that now it was to be taken as a definition of the protocol. For better or for worse it had been "cast in concrete", weaknesses and all. So it is not unreasonable to inquire whether as a **final definition** it is satisfactory under the criteria of Software Engineering; after all, the users of this system may be relying upon it for a critical operation.

Conclusions

The overwhelming conclusion is that the attempt to specify this process in a formal manner clearly revealed weaknesses that were opaque in the English. Also the task of defining the implementation needs for a node would have been much easier if the overall specification had been created in a formal manner in the first place. By using **Z** the specification of the individual processes was quickly achieved. However, as suspected from its state based nature, **Z** was not able to capture the

complete configuration, both sequential and parallel, of the individual processes as easily. Indeed it was decided, again purely for clarity since Parallelism can be expressed in **Z** [7], not to attempt the complete configuration. For this, languages currently under development like CSP and its timed variant are needed which are process algebra orientated and can easily handle parallelism. However this should not detract from the great gains in conceptual clarity which arose from using mature **Z**.

Acknowledgements

I thank Tim Dean, Gill Randell and Ruaridh Macdonald for their constructive comments during the preparation of this report.

References

[1] Communicating Sequential Processes, C.A.R. Hoare, Prentice-Hall International, 1985.

[2] Specification & Proof in Real-time Systems, J. Davies, D.Phil Dissertation, Oxford University, January 1991.

[3] The **Z** Notation A Reference Manual, J.M. Spivey, Prentice Hall International, 1989.

[4] DDN Protocol Handbook, vol 3, DDN Network Information Center, SRI International, December 1985.

[5] CCITT recommendation Z100, Nov 1988.

[6] CCITT recommendations X208 & X209.

[7] Parallel Refinement in **Z**, J.C.P. Woodcock, Oxford University, Private Communication

Appendix A

Network Working Group
Request For Comments: 826

David C. Plummer
(DCP@MIT-MC)
November 1982

An Ethernet Address Resolution Protocol
-- or --
Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware

Abstract

The implementation of protocol P on a sending host S decides, through protocol P's routing mechanism, that it wants to transmit to a target host T located some place on a connected piece of 10Mbit Ethernet cable. To actually transmit the Ethernet packet a 48.bit Ethernet address must be generated. The addresses of hosts within protocol P are not always compatible with the corresponding Ethernet address (being different lengths or values). Presented here is a protocol that allows dynamic distribution of the information needed to build tables to translate an address A in protocol P's address space into a 48.bit Ethernet address.

Generalizations have been made which allow the protocol to be used for non-10Mbit Ethernet hardware. Some packet radio networks are examples of such hardware.

The protocol proposed here is the result of a great deal of discussion with several other people, most notably J. Noel Chiappa, Yogen Dalal, and James E. Kulp, and helpful comments from David Moon.

[The purpose of this RFC is to present a method of Converting Protocol Addresses (e.g., IP addresses) to Local Network Addresses (e.g., Ethernet addresses). This is a issue of general concern in the ARPA Internet community at this time. The method proposed here is presented for your consideration and comment. This is not the specification of a Internet Standard.]

Notes:

This protocol was originally designed for the DEC/Intel/Xerox 10Mbit Ethernet. It has been generalized to allow it to be used for other types of networks. Much of the discussion will be directed toward the 10Mbit Ethernet. Generalizations, where applicable, will follow the Ethernet-specific discussion.

DOD Internet Protocol will be referred to as Internet.

Numbers here are in the Ethernet standard, which is high byte first. This is the opposite of the byte addressing of machines such as PDP-11s and VAXes. Therefore, special care must be taken with the opcode field (ar$op) described below.

An agreed upon authority is needed to manage hardware name space values (see below). Until an official authority exists, requests should be submitted to

David C. Plummer
Symbolics, Inc.
243 Vassar Street
Cambridge, Massachusetts 02139

Alternatively, network mail can be sent to DCP@MIT-MC.

The Problem:

The world is a jungle in general, and the networking game contributes many animals. At nearly every layer of a network architecture there are several potential protocols that could be used. For example, at a high level, there is TELNET and SUPDUP for remote login. Somewhere below that there is a reliable byte stream protocol, which might be CHAOS protocol, DOD TCP, Xerox BSP or DECnet. Even closer to the hardware is the logical transport layer, which might be CHAOS, DOD Internet, Xerox PUP, or DECnet. The 10Mbit Ethernet allows all of these protocols (and more) to coexist on a single cable by means of a type field in the Ethernet packet header. However, the 10Mbit Ethernet requires 48.bit addresses on the physical cable, yet most protocol addresses are not 48.bits long, nor do they necessarily have any relationship to the 48.bit Ethernet address of the hardware. For example, CHAOS addresses are 16.bits, DOD Internet addresses are 32.bits, and Xerox PUP addresses are 8.bits. A protocol is needed to dynamically distribute the correspondences between a <protocol, address> pair and a 48.bit Ethernet address.

Motivation:

Use of the 10Mbit Ethernet is increasing as more manufacturers

supply interfaces that conform to the specification published by DEC, Intel and Xerox. With this increasing availability, more and more software is being written for these interfaces. There are two alternatives: (1) Every implementor invents his/her own method to do some form of address resolution, or (2) every implementor uses a standard so that his/her code can be distributed to other systems without need for modification. This proposal attempts to set the standard.

Definitions:

Define the following for referring to the values put in the TYPE field of the Ethernet packet header:

ether_type$XEROX_PUP,
ether_type$DOD_INTERNET,
ether_type$CHAOS,

and a new one:

ether_type$ADDRESS_RESOLUTION.

Also define the following values (to be discussed later):

ares_op$REQUEST (= 1, high byte transmitted first) and
ares_op$REPLY (= 2),

and

ares_hrd$Ethernet (= 1).

Packet format:

To communicate mappings from <protocol, address> pairs to 48.bit Ethernet addresses, a packet format that embodies the Address Resolution protocol is needed. The format of the packet follows.

Ethernet transmission layer (not necessarily accessible to the user):
48.bit: Ethernet address of destination
48.bit: Ethernet address of sender
16.bit: Protocol type = ether_type$ADDRESS_RESOLUTION
Ethernet packet data:
16.bit: (ar$hrd) Hardware address space (e.g., Ethernet, Packet Radio Net.)
16.bit: (ar$pro) Protocol address space. For Ethernet hardware, this is from the set of type fields ether_typ$<protocol>.
8.bit: (ar$hln) byte length of each hardware address
8.bit: (ar$pln) byte length of each protocol address
16.bit: (ar$op) opcode (ares_op$REQUEST | ares_op$REPLY)
nbytes: (ar$sha) Hardware address of sender of this packet, n from the ar$hln field.
mbytes: (ar$spa) Protocol address of sender of this packet, m from the ar$pln field.
nbytes: (ar$tha) Hardware address of target of this packet (if known).

mbytes: (ar$tpa) Protocol address of target.

Packet Generation:

As a packet is sent down through the network layers, routing determines the protocol address of the next hop for the packet and on which piece of hardware it expects to find the station with the immediate target protocol address. In the case of the 10Mbit Ethernet, address resolution is needed and some lower layer (probably the hardware driver) must consult the Address Resolution module (perhaps implemented in the Ethernet support module) to convert the <protocol type, target protocol address> pair to a 48.bit Ethernet address. The Address Resolution module tries to find this pair in a table. If it finds the pair, it gives the corresponding 48.bit Ethernet address back to the caller (hardware driver) which then transmits the packet. If it does not, it probably informs the caller that it is throwing the packet away (on the assumption the packet will be retransmitted by a higher network layer), and generates an Ethernet packet with a type field of ether_type$ADDRESS_RESOLUTION. The Address Resolution module then sets the ar$hrd field to ares_hrd$Ethernet, ar$pro to the protocol type that is being resolved, ar$hln to 6 (the number of bytes in a 48.bit Ethernet address), ar$pln to the length of an address in that protocol, ar$op to ares_op$REQUEST, ar$sha with the 48.bit ethernet address of itself, ar$spa with the protocol address of itself, and ar$tpa with the protocol address of the machine that is trying to be accessed. It does not set ar$tha to anything in particular, because it is this value that it is trying to determine. It could set ar$tha to the broadcast address for the hardware (all ones in the case of the 10Mbit Ethernet) if that makes it convenient for some aspect of the implementation. It then causes this packet to be broadcast to all stations on the Ethernet cable originally determined by the routing mechanism.

Packet Reception:

When an address resolution packet is received, the receiving Ethernet module gives the packet to the Address Resolution module which goes through an algorithm similar to the following. Negative conditionals indicate an end of processing and a discarding of the packet.

?Do I have the hardware type in ar$hrd?
Yes: (almost definitely)
  [optionally check the hardware length ar$hln]
  ?Do I speak the protocol in ar$pro?
  Yes:

```
[optionally check the protocol length ar$pln]
Merge_flag := false
If the pair <protocol type, sender protocol address> is
   already in my translation table, update the sender
   hardware address field of the entry with the new
   information in the packet and set Merge_flag to true.
?Am I the target protocol address?
Yes:
  If Merge_flag is false, add the triplet <protocol type,
     sender protocol address, sender hardware address> to
    the translation table.
  ?Is the opcode ares_op$REQUEST? (NOW look at the opcode!!)
  Yes:
    Swap hardware and protocol fields, putting the local
       hardware and protocol addresses in the sender fields.
    Set the ar$op field to ares_op$REPLY
    Send the packet to the (new) target hardware address on
       the same hardware on which the request was received.
```

Notice that the <protocol type, sender protocol address, sender hardware address> triplet is merged into the table before the opcode is looked at. This is on the assumption that communcation is bidirectional; if A has some reason to talk to B, then B will probably have some reason to talk to A. Notice also that if an entry already exists for the <protocol type, sender protocol address> pair, then the new hardware address supersedes the old one. Related Issues gives some motivation for this.

Generalization: The ar$hrd and ar$hln fields allow this protocol and packet format to be used for non-10Mbit Ethernets. For the 10Mbit Ethernet <arhrd, arhln> takes on the value <1, 6>. For other hardware networks, the ar$pro field may no longer correspond to the Ethernet type field, but it should be associated with the protocol whose address resolution is being sought.

Why is it done this way??

Periodic broadcasting is definitely not desired. Imagine 100 workstations on a single Ethernet, each broadcasting address resolution information once per 10 minutes (as one possible set of parameters). This is one packet every 6 seconds. This is almost reasonable, but what use is it? The workstations aren't generally going to be talking to each other (and therefore have 100 useless entries in a table); they will be mainly talking to a mainframe, file server or bridge, but only to a small number of other workstations (for interactive conversations, for example). The protocol described in this paper distributes information as it is needed, and only once (probably) per boot of a machine.

This format does not allow for more than one resolution to be done in the same packet. This is for simplicity. If things were

multiplexed the packet format would be considerably harder to digest, and much of the information could be gratuitous. Think of a bridge that talks four protocols telling a workstation all four protocol addresses, three of which the workstation will probably never use.

This format allows the packet buffer to be reused if a reply is generated; a reply has the same length as a request, and several of the fields are the same.

The value of the hardware field (ar$hrd) is taken from a list for this purpose. Currently the only defined value is for the 10Mbit Ethernet (ares_hrd$Ethernet = 1). There has been talk of using this protocol for Packet Radio Networks as well, and this will require another value as will other future hardware mediums that wish to use this protocol.

For the 10Mbit Ethernet, the value in the protocol field (ar$pro) is taken from the set ether_type$. This is a natural reuse of the assigned protocol types. Combining this with the opcode (ar$op) would effectively halve the number of protocols that can be resolved under this protocol and would make a monitor/debugger more complex (see Network Monitoring and Debugging below). It is hoped that we will never see 32768 protocols, but Murphy made some laws which don't allow us to make this assumption.

In theory, the length fields (ar$hln and ar$pln) are redundant, since the length of a protocol address should be determined by the hardware type (found in ar$hrd) and the protocol type (found in ar$pro). It is included for optional consistency checking, and for network monitoring and debugging (see below).

The opcode is to determine if this is a request (which may cause a reply) or a reply to a previous request. 16 bits for this is overkill, but a flag (field) is needed.

The sender hardware address and sender protocol address are absolutely necessary. It is these fields that get put in a translation table.

The target protocol address is necessary in the request form of the packet so that a machine can determine whether or not to enter the sender information in a table or to send a reply. It is not necessarily needed in the reply form if one assumes a reply is only provoked by a request. It is included for completeness, network monitoring, and to simplify the suggested processing algorithm described above (which does not look at the opcode until AFTER putting the sender information in a table).

The target hardware address is included for completeness and network monitoring. It has no meaning in the request form, since it is this number that the machine is requesting. Its meaning in the reply form is the address of the machine making the request.

In some implementations (which do not get to look at the 14.byte ethernet header, for example) this may save some register shuffling or stack space by sending this field to the hardware driver as the hardware destination address of the packet. There are no padding bytes between addresses. The packet data should be viewed as a byte stream in which only 3 byte pairs are defined to be words (arhrd, arpro and ar$op) which are sent most significant byte first (Ethernet/PDP-10 byte style).

Network monitoring and debugging:

The above Address Resolution protocol allows a machine to gain knowledge about the higher level protocol activity (e.g., CHAOS, Internet, PUP, DECnet) on an Ethernet cable. It can determine which Ethernet protocol type fields are in use (by value) and the protocol addresses within each protocol type. In fact, it is not necessary for the monitor to speak any of the higher level protocols involved. It goes something like this:

When a monitor receives an Address Resolution packet, it always enters the <protocol type, sender protocol address, sender hardware address> in a table. It can determine the length of the hardware and protocol address from the ar$hln and ar$pln fields of the packet. If the opcode is a REPLY the monitor can then throw the packet away. If the opcode is a REQUEST and the target protocol address matches the protocol address of the monitor, the monitor sends a REPLY as it normally would. The monitor will only get one mapping this way, since the REPLY to the REQUEST will be sent directly to the requesting host. The monitor could try sending its own REQUEST, but this could get two monitors into a REQUEST sending loop, and care must be taken.

Because the protocol and opcode are not combined into one field, the monitor does not need to know which request opcode is associated with which reply opcode for the same higher level protocol. The length fields should also give enough information to enable it to "parse" a protocol addresses, although it has no knowledge of what the protocol addresses mean.

A working implementation of the Address Resolution protocol can also be used to debug a non-working implementation. Presumably a hardware driver will successfully broadcast a packet with Ethernet type field of ether_type$ADDRESS_RESOLUTION. The format of the packet may not be totally correct, because initial implementations may have bugs, and table management may be slightly tricky. Because requests are broadcast a monitor will receive the packet and can display it for debugging if desired.

An Example:

Let there exist machines X and Y that are on the same 10Mbit Ethernet cable. They have Ethernet address EA(X) and EA(Y) and DOD Internet addresses IPA(X) and IPA(Y). Let the Ethernet type of Internet be ET(IP). Machine X has just been started, and sooner or later wants to send an Internet packet to machine Y on the same cable. X knows that it wants to send to IPA(Y) and tells the hardware driver (here an Ethernet driver) IPA(Y). The driver consults the Address Resolution module to convert <ET(IP), IPA(Y)> into a 48.bit Ethernet address, but because X was just started, it does not have this information. It throws the Internet packet away and instead creates an ADDRESS RESOLUTION packet with

(ar$hrd) = ares_hrd$Ethernet
(ar$pro) = ET(IP)
(ar$hln) = length(EA(X))
(ar$pln) = length(IPA(X))
(ar$op) = ares_op$REQUEST
(ar$sha) = EA(X)
(ar$spa) = IPA(X)
(ar$tha) = don't care
(ar$tpa) = IPA(Y)

and broadcasts this packet to everybody on the cable.

Machine Y gets this packet, and determines that it understands the hardware type (Ethernet), that it speaks the indicated protocol (Internet) and that the packet is for it ((ar$tpa)=IPA(Y)). It enters (probably replacing any existing entry) the information that <ET(IP), IPA(X)> maps to EA(X). It then notices that it is a request, so it swaps fields, putting EA(Y) in the new sender Ethernet address field (ar$sha), sets the opcode to reply, and sends the packet directly (not broadcast) to EA(X). At this point Y knows how to send to X, but X still doesn't know how to send to Y.

Machine X gets the reply packet from Y, forms the map from <ET(IP), IPA(Y)> to EA(Y), notices the packet is a reply and throws it away. The next time X's Internet module tries to send a packet to Y on the Ethernet, the translation will succeed, and the packet will (hopefully) arrive. If Y's Internet module then wants to talk to X, this will also succeed since Y has remembered the information from X's request for Address Resolution.

Related issue:

It may be desirable to have table aging and/or timeouts. The implementation of these is outside the scope of this protocol. Here is a more detailed description (thanks to MOON@SCRC@MIT-MC).

If a host moves, any connections initiated by that host will work, assuming its own address resolution table is cleared when it moves. However, connections initiated to it by other hosts

will have no particular reason to know to discard their old address. However, 48.bit Ethernet addresses are supposed to be unique and fixed for all time, so they shouldn't change. A host could "move" if a host name (and address in some other protocol) were reassigned to a different physical piece of hardware. Also, as we know from experience, there is always the danger of incorrect routing information accidentally getting transmitted through hardware or software error; it should not be allowed to persist forever. Perhaps failure to initiate a connection should inform the Address Resolution module to delete the information on the basis that the host is not reachable, possibly because it is down or the old translation is no longer valid. Or perhaps receiving of a packet from a host should reset a timeout in the address resolution entry used for transmitting packets to that host; if no packets are received from a host for a suitable length of time, the address resolution entry is forgotten. This may cause extra overhead to scan the table for each incoming packet. Perhaps a hash or index can make this faster.

The suggested algorithm for receiving address resolution packets tries to lessen the time it takes for recovery if a host does move. Recall that if the <protocol type, sender protocol address> is already in the translation table, then the sender hardware address supersedes the existing entry. Therefore, on a perfect Ethernet where a broadcast REQUEST reaches all stations on the cable, each station will be get the new hardware address.

Another alternative is to have a daemon perform the timeouts. After a suitable time, the daemon considers removing an entry. It first sends (with a small number of retransmissions if needed)

an address resolution packet with opcode REQUEST directly to the Ethernet address in the table. If a REPLY is not seen in a short amount of time, the entry is deleted. The request is sent directly so as not to bother every station on the Ethernet. Just forgetting entries will likely cause useful information to be forgotten, which must be regained.

Since hosts don't transmit information about anyone other than themselves, rebooting a host will cause its address mapping table to be up to date. Bad information can't persist forever by being passed around from machine to machine; the only bad information that can exist is in a machine that doesn't know that some other machine has changed its 48.bit Ethernet address. Perhaps manually resetting (or clearing) the address mapping table will suffice.

This issue clearly needs more thought if it is believed to be important. It is caused by any address resolution-like protocol.

Appendix B

Z specification of RFC826

The given types

The specification document does not give the number of protocols, the detailed internal structure of protocol addresses, the types of hardware and the hardware addresses. So it is appropriate to treat them as sets of given elements. Hence the given data are :

the types of protocols, the addresses in the formats of those protocols, the types of hardware and the hardware addresses of the systems.

```
[PROTOCOLS, P_ADDRESSES,
        HARDWARE_TYPES, HARDWARE_ADDRESSES]
```

Ethernet addresses are defined as a subset of possible hardware addresses. The operation requires an explicit protocol sort called address_resolution,

```
ether_addresses : ℙ HARDWARE_ADDRESSES
address_resolution : PROTOCOLS
```

..... and a hardware address for broadcast messages.

```
broadcast : ether_addresses
```

The address resolution module requires two opcodes.

```
OPCODE ::= REQUEST | REPLY
```

Node characteristics

Each node has a list of node characteristics which are encapsulated in a single schema. Here the information about the protocols the node supports and the corresponding address is stored as a partial injection; partial because not all protocols may be supported, and a function because different protocols could use the same address for the same node. Note that the node hardware address is drawn from the given hardware addresses and not the Ether subset. This is in the spirit of the generalisation attempt alluded to in the specification document

```
Node_data
node_protocols: PROTOCOLS ⇸ P_ADDRESSES
node_hardware_type : HARDWARE_TYPES
node_hardware_address : HARDWARE_ADDRESSES
```

The address table

Each node has an address table which also incorporates the node's own characteristics.

The address lookup table, which is at the heart of this protocol, is modelled as a partial function. The mapping is partial because not all possible protocols and associated addresses will be stored on the table, and a function because more than one protocol type and address can correspond to a given hardware address.

It is convienient to consider two tables. The first is the table of the node being asked the address question.

```
Requestor_table
address_table: PROTOCOLS × P_ADDRESSES
                    ⇸ HARDWARE_ADDRESSES
Node_data
```

The second is the table of another node on the Ethernet which may have the desired information.

```
Recipient_table
address_table: PROTOCOLS × P_ADDRESSES
                    ⇸ HARDWARE_ADDRESSES
Node_data
```

Local table search

This defines a successful local table search by the (potential) requestor. If the desired protocol type and address (treated as inputs with the ? suffix) are known then they will be present in the domain of the table. The desired hardware address (treated as output with the ! suffix) will be the appropriate element of the table's co-domain.

```
Successful_local_find
Ξ Requestor_table
wanted_protocol_type? : PROTOCOLS
wanted_protocol_address?: P_ADDRESSES
hardware_address! : HARDWARE_ADDRESSES
---
( wanted_protocol_type?, wanted_protocol_address? ) ∈
                              dom address_table
hardware_address! =
            address_table ( wanted_protocol_type?,
                            wanted_protocol_address? )
```

The Ethernet packet

The structure of the relevant parts (the two fields for the byte lengths of the addresses are omitted) of an address resolution Ethernet packet is represented as a schema. Note that this does not give the explicit structure of the packet (ie the ordered bit string), rather it concentrates on the crucial point that the elements of the schema are distinct.

```
Ether_packet
eadd_dest,eadd_sender : ETHER_ADDRESSES
data_protocol_format: PROTOCOLS
hardware_address_space : HARDWARE_TYPES
protocol_address_space : PROTOCOLS
op : OPCODE
hardware_address_sender : HARDWARE_ADDRESSES
protocol_address_sender : P_ADDRESSES
hardware_address_target : HARDWARE_ADDRESSES
protocol_address_target : P_ADDRESSES
```

Send request

The Send_request schema specifies the values of a request to all the other systems on the Ethernet. This asks for the Ethernet address of the sought after node whose Protocol type and address have been supplied. The sending of a request is specified by expressing the state of the etherpacket after the operation, which involved setting the packet type to be an address resolution request, providing the ether address of the requestor and the protocol details of the sought after node. Note: an extra precondition test has had to be added which requires the hardware address of the sender to be an Ethernet address. This arises from the partial generalisation inconsistency mentioned in the main text.

```
Send_request
Ξ Requestor_table
wanted_protocol_type? : PROTOCOLS
wanted_protocol_address? : P_ADDRESSES
Ether_packet!
---
node_hardware_address ∈ ether_addresses
eadd_sender! = node_hardware_address
eadd_dest! = broadcast
data_protocol_format! = address_resolution
hardware_address_space! = node_hardware_type
protocol_address_space! = wanted_protocol_type?
op! = REQUEST
hardware_address_sender! = node_hardware_address
protocol_address_sender! =
          node_protocols ( wanted_protocol_type? )
protocol_address_target! = wanted_protocol_address?
```

Receive request

The receive_request schema describes the reception of the address resolution packet by all the recipient systems on the Ethernet. Here the conditions for a successful (ie valid) receive (for this RFC protocol) at a given Ethernet address are specified. These are tests on the Ethernet packet contents to see if either the Ethernet packet was a broadcast packet or it was for this particular node (Note: this last test is implied in the specification document but not explicitly stated). Also further tests check that the packet is an address resolution packet, that it is for the hardware type of node and the protocol type that the node supports.

```
Receive_request
Ξ Recipient_table
Ether_packet?
─────────────────────────────────────────
eadd_dest? = broadcast ∨
     eadd_dest? = node_hardware_address
data_protocol_format? = address_resolution
hardware_address_space? = node_hardware_type
protocol_address_space? ∈
                    dom node_protocols
```

Existing entry check

Here the recipient node tests for knowledge of the sender by seeing if the sender's protocol details, ie type and address, are in the node's own look up table. If they are then the local table entry is updated by overwriting the previous entry. Note: this update is done even if the node is not the one being sought.

```
Existing_entry_check
Δ Recipient_table
Ether_packet?
─────────────────────────────────────────────
( protocol_address_space?, protocol_address_sender? )
                              ∈ dom address_table
address_table' = address_table ⊕
 { (protocol_address_space?,protocol_address_sender?)
                          ↦ hardware_address_sender? }
Θ Node_data' = Θ Node_data
```

Add new entry

Here if the sender is not known to the recipient and the node being sought is indeed that node then its table is updated with the sender's details.

```
Add_new_entry
Δ Recipient_table
Ether_packet?
─────────────────────────────────────────────
( protocol_address_space?, protocol_address_sender? )
                              ∉ dom address_table
protocol_address_target? =
            node_protocols( protocol_address_space?)
address_table' = address_table ∪
  {(protcol_address_space?,protocol_address_sender?)
                          ↦ hardware_address_sender? }
Θ Node_data' = Θ Node_data
```

Reply

The last action of the recipient, is to send the desired information back to the requestor. Here the condition to be met is that the original action in the packet was a request and that the action is then set to be a reply with the addresses being assigned to the appropriate parts of the ether packet.

```
Reply
Ξ Recipient_table
Ether_packet?
Ether_packet!
-----------------------------------------------------
op? = REQUEST
node_hardware_address ∈ ether_addresses
eadd_sender! = node_hardware_address
eadd_dest! = hardware_address_sender?
data_protocol_format! = data_protocol_format?
hardware_address_space! = hardware_address_space?
protocol_address_space! = protocol_address_space?
op! = REPLY
hardware_address_sender! = node_hardware_address
protocol_address_sender! =
          node_protocols ( protocol_address_space?)
hardware_address_target! = hardware_address_sender?
protocol_address_target! = protocol_address_sender?
```

Update table

In the last stage the requestor receives the reply and updates its local address table. The condition tested is that the packet is the reply of an address resolution packet. The address table is then updated from the appropriate data parts of the packet.

```
Update_table
Δ Requestor_table
Ether_packet?
-----------------------------------------------
data_protocol_format? = address_resolution
op? = REPLY
address_table' = address_table ⊔
 { ( protcol_address_space?,protocol_address_sender?)
    ↦ hardware_address_sender }
Θ Node_data' = Θ Node_data
```

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) .....

Overall security classification of sheet .....UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
REPORT 91004	JUNE	1991

Originators Name and Location

RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title

FROM ENGLISH TO Z, A REVERSE SPECIFICATION PROCESS

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period
Project Number	**Other References**

Authors	Pagination and Ref
GIESS, S C	27

Abstract

On an exercise to translate the communication protocol specification RFC 826, 'The address resolution protocol', already written in English, into the mathematical specification language **Z**.

Abstract Classification (U,R,C or S)

U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)

UNLIMITED